

# Automatic Generation of Markov Chain Usage Models from Real-time Software UML Models

Jiong Yan<sup>†</sup>, Ji Wang<sup>‡</sup>, and Huo-wang Chen

National Laboratory for Parallel and Distributed Processing, Changsha, P.R. China

<sup>†</sup>E-mail: [yanjiong7172@vip.sina.com](mailto:yanjiong7172@vip.sina.com)

<sup>‡</sup>E-mail: [jiwang@mail.edu.cn](mailto:jiwang@mail.edu.cn)

## Abstract

*The paper concerns automatic generation of usage models from real-time software UML models. Firstly, we define the reasonably constrained real-time software UML artifacts, which include use case diagrams, timed sequence diagrams and the execution probability of each sequence diagram in its associated use case. Secondly, the paper presents a method that derives the software usage model from the constrained UML artifacts. The method elicits the messages associated with the objects under testing and their occurrence probabilities to generate the usage model of each use case. Timing constraints in sequence diagrams are considered during usage model generation. Then the usage models of use cases are integrated into the software usage model by utilizing the execution sequence relations between use cases. The usage models can be used to generate real-time software statistical test cases and facilitate real-time software statistical testing.*

## 1. Introduction

Recently software reliability is widely studied because the failures of the software used in high dependable systems are often disastrous. Software statistical testing is concerned with testing the entire software systems based on their usage models, and is also called reliability testing, for it generates test cases from the software usage models and thus can be used to estimate software reliability [13]. As the verification technique of Cleanroom method, statistical testing plays a key role in the verification of high dependable software. The Cleanroom method describes software usage model as a Markov chain, from which test cases are derived [13].

While object-oriented software development has been the defacto standard software development

method, traditional testing methods are still widely used in software testing. Because UML has been the standard modeling language in object-oriented software development, it is desired to combine UML-based software development with statistical testing. UML-based statistical testing can not only support test cases derivation but also test planning in the early stages of software development.

There are many studies on use case and scenario base software testing [2, 5, 7]. Whereas, there are still not many research works on statistical testing for UML-based development. [19] derives use cases' Markov chain usage models and then integrates the use cases' usage models to generate software usage model, but does not consider time related issues. [15] presents an approach to generating system-level test cases based on use cases and refined by state diagrams, and transforms these models into usage models to describe the system behavior and usage.

Most projects modeled with UML produce use case diagrams, sequence diagrams associated with each use case, and class diagrams [9]. Whereas these artifacts are often not testable for they haven't provided sufficient information to allow automatic generation of test cases. Therefore, in order to support statistical testing it is necessary to introduce testability constraints to UML artifacts. Because of the complexity of high dependable software, large numbers of test cases are needed in statistical testing. Thus, those testing methods requiring frequent and complex manual tasks are not likely to be widely used.

This paper gives our attempt to facilitate statistical testing of the real-time software. We present a method that derives the real-time software Markov chain usage models from reasonably constrained UML analysis artifacts.

The rest of the paper is organized as follows. Section 2 formalizes UML artifacts with timing constraints and statistical testing constraints; those constraints are indispensable to UML based real-time software

statistical testing. Section 3 studies the approach to generating Markov chain usage model from the constrained UML artifacts in detail. Section 4 explains statistical test cases derivation and the architecture of supporting tool. The conclusions are drawn in Section 5.

## 2. Formalizing UML artifacts

This section formally defines UML artifacts with statistical testing constraints and real-time constraints.

### 2.1. Scenarios and timing constraints

In the UML based software development, use cases describe high-level functionalities of software, and scenarios describe how system components and actors interact in order to provide a system level functionality. The notations for scenarios are message sequence charts (MSCs) [11] and UML sequence diagram [16]. There are two fundamentally different approaches to scenario semantics, which impact the role of scenario-based specification within the development process. The first category considers that scenarios describe high-level design of system components, thus takes on a design-oriented perspective and uses state machine semantics [10]. The second interpretation of scenarios considers them system functionality description, which is more suitable in an early requirements view and is taken by approaches using semantics based on partial ordering of events [4].

There are essentially four classes of syntactic constructs to express timing constraints in scenarios [1]:

**Timers.** A timer can be used to express a maximal delay between two events in one process (or lifeline, in the notation of UML).

**Delay intervals.** They can express time intervals between two consecutive events in a process.

**Drawing rules and timing markers.** They are used in UML to express timing constraints. For example, a downward slanted message arrow indicates a message delivery delay. Timing markers are Boolean expressions attached to the diagram, which are more expressive than timers and delays.

In order to support real-time software development, OMG extends UML with a framework for representing time and time related mechanisms [14]. The extension provides TimeValue, TimeInterval, clock, timer, TimedAction, TimedEvent, TimedStimulus, Timeout, Delay, ClockInterrupt, TimeService, and <<RT>> stereotypes to support real-time software modeling. The timed scenarios can be used to validate timing assignment and verify timing consistency [3, 12].

### 2.2. Formalizing use cases and scenarios

We first formalize sequence diagrams based on partial ordering of events. A sequence diagram describes a finite set of interactions between a set of objects and has two dimensions: the vertical dimension represents time and the horizontal dimension represents different instances, which are objects that participates the interactions. For each instance there is a vertical line, called lifeline, on which a local visual order of all the events belonging to the instance are defined. Each event is either a send or a receive event, and belongs to one specific instance. For each send event, there exists a matching receive event, and vice versa. Thus, each message is associated with one send event and one receive event. Actors represent the environment.

A sequence diagram  $m$  is defined as a 4-tuple:

$$m = (inst_m, dom_m, message_m, TmConstraints_m)$$

where

- $inst_m$  is a finite set of instances.
- $dom_m$  is a finite set  $S$  of send events and a finite set  $R$  of receive events. Each event is labeled with a natural number and is associated with a time mark that specifies the event occurrence time. Accordingly, each instance  $i$  is associated with a finite number of discrete events with time marks. Thus,  

$$dom_m = \{ \langle i, e, t_e, x \rangle \mid i \in inst_m, e \in \text{The events on object } i\text{'s lifeline}, t_e \text{ is occurrence time of } e, x = ! \text{ if } e \text{ is a send event, otherwise } x = ? \}$$
- $message_m$  is a finite set of messages. A message is labeled by a message id with its operation name and corresponding parameter list. For a sequence diagram  $m$ , the association between events and messages is given by function  $msg_m$ :  

$$msg_m : dom_m \rightarrow \mathcal{P}(message_m)$$
- $TmConstraints_m$  is the set of timing constraints. A timing constraint is a 4-tuple:  $(TC, TmViolationHandler, TmViolationPost, TmViolationProb)$ , where

1.  $TC$  is a timing constraint defined as  $t_e + a < t_{e'} < t_e + b$ , where  $t_e$  and  $t_{e'}$  are occurrence time of event  $e$  and  $e'$ ,  $a$  and  $b$  are delays, which means that event  $e'$  must occur in the time interval  $(a, b)$  after  $e$  occurs, otherwise time violation occurs. Notice that  $e$  and  $e'$  are events on the same instance and  $e'$  is the consecutive event of  $e$ . We call that event  $e'$  is attached with a timing constraint. We assume that the timing constraints are mutually independent, event  $e'$  can only occur on discrete time point in the time interval  $(a, b)$  after  $e$  occurs.

2. *TmViolationHandler* is the exception handler that processes the time violation.
3. *TmViolationPost* is the execution postcondition of time violation handler.
4. *TmViolationProb* is the time violation probability of *TC* in sequence diagram *m*.

The functions *source* and *target* define the source instance and target instance of a message:

$source, target : message_m \rightarrow inst_m \cup \{E\}$ .

where  $\langle i, e, t_e, x \rangle \in dom_m$ ,  $m\_id \in message_m$  and *E* is the environment:

$$source : \mu \rightarrow \begin{cases} i : \mu = m\_id \in msg_m(\langle i, e, t_e, ! \rangle) \\ E : otherwise \end{cases}$$

$$target : \mu \rightarrow \begin{cases} i : \mu = m\_id \in msg_m(\langle i, e, t_e, ? \rangle) \\ E : otherwise \end{cases}$$

In the following definitions, we assume that  $x, x' \text{ and } x'' \in \{!, ?\}$ .

For a sequence diagram *m*, the relation between events is  $\leq \subseteq dom_m \times dom_m$ , which is defined by:

1.  $\langle i, e, t_e, x \rangle \leq \langle i, e+1, t_{e+1}, x' \rangle$ ;
2.  $m\_id \in msg_m(\langle i, e, t_e, ! \rangle)$ ,  $m\_id \in msg_m(\langle i', e', t_{e'}, ? \rangle) \Rightarrow \langle i, e, t_e, ! \rangle \leq \langle i', e', t_{e'}, ? \rangle$ ;
3. Transitive closure induced by 1 and 2.

The function *msg<sub>m</sub>* must maintain the following constraints for consistency.

1. Each message is unique in the sequence diagram:

$\forall \langle i, e, t_e, x \rangle \text{ and } \langle i', e', t_{e'}, x \rangle \in dom_m$ :

$m\_id \in msg_m(\langle i, e, t_e, x \rangle) \wedge$

$m\_id \in msg_m(\langle i', e', t_{e'}, x \rangle)$

$\Rightarrow \langle i, e, t_e, x \rangle = \langle i', e', t_{e'}, x \rangle$

2. It is not possible to receive more than one message on each event, or receive a message and send another message on the same event:

$\forall \langle i, e, t_e, x \rangle \in dom_m, m\_id \in msg_m(\langle i, e, t_e, x \rangle)$ :  
 $\#(msg_m(\langle i, e, t_e, x \rangle)) > 1 \Rightarrow x = !$

The relation between two messages in a sequence diagram *m* is defined as  $\prec \subseteq message_m \times message_m$ , which satisfies:

$\forall m\_id, m\_id' \in message_m$ ,

$\exists \langle i, e, t_e, x \rangle \text{ and } \langle i', e', t_{e'}, x' \rangle \in dom_m$ :

$m\_id \in msg_m(\langle i, e, t_e, x \rangle) \wedge$

$m\_id' \in msg_m(\langle i', e', t_{e'}, x' \rangle) \wedge$

$\langle i, e, t_e, x \rangle \leq \langle i', e', t_{e'}, x' \rangle$

$\Rightarrow m\_id \prec m\_id'$

Use cases not only have <<extend>> and <<include>> relationships but also execution sequence relations which reflect the business process the software supports [6]. Software statistical testing must consider the execution sequence relations because different use

case execution sequences may trigger different failures. We can represent execution sequence relations between use cases by an activity diagram [8]. In such a diagram, vertices are use cases and the edges are execution sequence relations between use cases. To execute a use case, specific preconditions must be satisfied. After use case's execution the software will enter specific postconditions, which determine which use case will be executed next. For each use case, the execution of different scenarios may cause the software enter different states. Thus, we attach a postcondition for each sequence diagram.

A use case *UC* can be defined as

$$UC = (pre_{uc}, SDSet_{uc})$$

where *SDSet<sub>uc</sub>* is the set of sequence diagrams associated with this use case, *pre<sub>uc</sub>* is the precondition set of the use case, specified with Object Constraint Language (OCL) [17].

Thus, a sequence diagram *m* with statistical testing constraints can be defined as

$$m = (inst_m, dom_m, msg_m, post_m, pf_m, TmConstraints_m)$$

where *inst<sub>m</sub>*, *dom<sub>m</sub>*, *msg<sub>m</sub>* and *TmConstraints<sub>m</sub>* are defined as above, *post<sub>m</sub>* is the execution postcondition of the sequence diagram *m*, specified with OCL, *pf<sub>m</sub>* is the execution probability of *m* in its associated use case. Thus, for all sequence diagrams *m* in a use case *U*,

$$\sum_{m \in U} pf_m = 1.$$

The execution sequence relation of use cases can be defined as:

$$UCExecRelation = (UCSet, s_{start}, s_{end}, V, \sum, seq)$$

where

- *UCSet* is the set of software use cases.
- *s<sub>start</sub>* is the initial node which represents the start of the software execution.
- *s<sub>end</sub>* is the final node that represents the end of the software execution.
- $V = \{s_{start}, s_{end}\} \cup UCSet$ .
- $\sum = \cup(pre_{uc} \cup post_m)$ , for all use cases and sequence diagrams.
- *seq* is the function that specifies the execution sequence relations between use cases, initial node and final node.

$$seq : V \times \sum \times (0, 1] \rightarrow V$$

A UML model is defined as:

$$system = (actors, UCSet, UCExecRelation)$$

where

- *actors* is the set of the software users.
- *UCSet* is the set of software use cases.
- *UCExecRelation* is the execution sequence relations of use cases.

### 2.3. A SCS case

[13] provides an example of a satellite communication system, which consists of ground control system (GCS), satellite control software (SCS), uplink site (UL) and downlink site (DL). We here address only the Satellite Control Software (SCS). Figure 1 is the use case diagram of SCS.

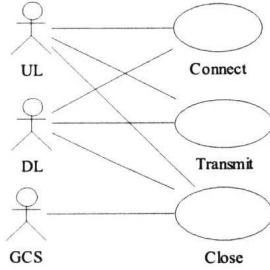


Figure 1. Use case diagram of SCS

The initial state of SCS is Idle. SCS must perform use case *Connect* before communicating. Successful connection makes SCS enter state Connected and can execute use case *Transmit*, while unsuccessful connection makes the SCS enter state Idle and thus terminates the execution of SCS. Transmission continues until UL sends all packets to SCS and SCS forwards all packets to DL. Having sent all packets, the UL will send a TC message to SCS and close the connection, SCS enters state Idle. Figure 2 describes the execution sequence relations between use cases of SCS with a UML activity diagram. In this activity diagram, the transition probability is 1 if it is not labeled.

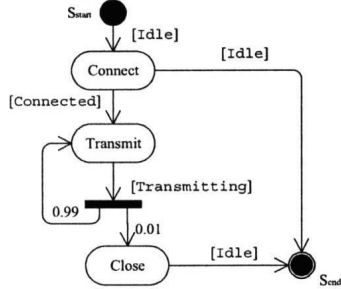


Figure 2. SCS use cases execution sequence relations

The SCS is defined as:

$SCS =_{df} (actors, UCSet, UCExecRelation)$ , where

$actors =_{df} (GCS, DL, UL)$

$UCSet =_{df} (Connect, Transmit, Close)$

$UCExecRelation$  is depicted in Figure 2.

The use cases are:

$Connect =_{df} (<Idle>, <m_{Connect, 1}, m_{Connect, 2}, m_{Connect, 3}>)$

$Transmit =_{df} (<Transmitting, Connected>,$

$<m_{Transmit, 1}, m_{Transmit, 2}>)$

$Close =_{df} (<Transmitting>, <m_{Close}>)$

Figure 3 to 8 depict the scenarios. Table 1 specifies the execution probability of each scenario in its associated use case, as well as the postcondition.

Table 1. Execution probability and postcondition of each scenario in its associated use case

		Connect	Transmit	Close
SD1	$pf_1$	0.8	0.8	1
	$post_1$	Connected	Transmitting	Idle
SD2	$pf_2$	0.1	0.2	-
	$post_2$	Idle	Transmitting	
SD3	$pf_3$	0.1	-	-
	$post_3$	Idle		

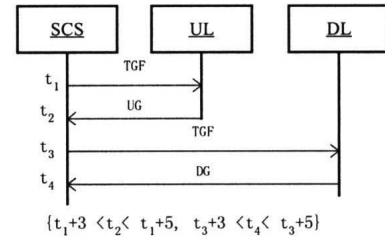


Figure 3. Scenario 1 of Connect

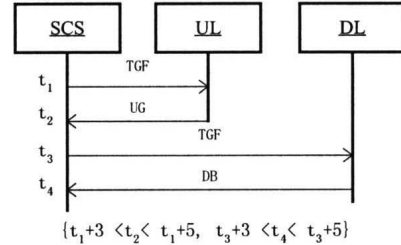


Figure 4. Scenario 2 of Connect

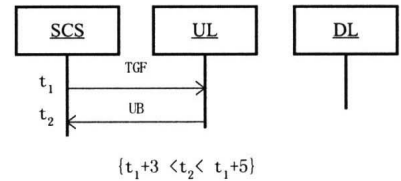


Figure 5. Scenario 3 of Connect

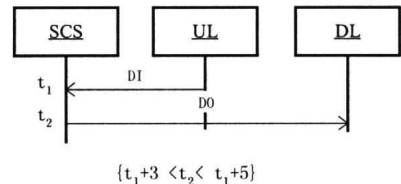


Figure 6. Scenario 1 of Transmit

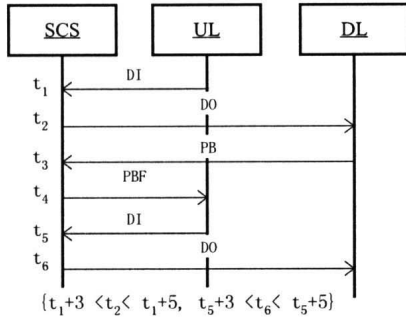


Figure 7. Scenario 2 of Transmit

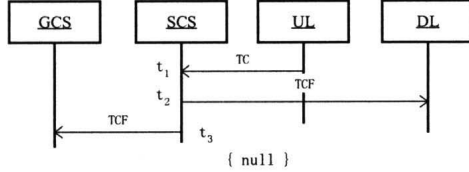


Figure 8. Scenario 1 of Close

We assume that the time violation probability of each timing constraint in use case *Connect* and *Transmit* is 0.01, the time violation handler is *TVProc* and the execution postcondition of time violation handler is *Idle*.

### 3. Deriving real-time software usage models from UML artifacts

This section presents a method that derives Markov chain usage model from UML artifacts.

#### 3.1. UML-based software statistical testing

UML-based statistical testing presented in this paper includes the following steps.

1. Adding statistical testing constraints and timing constraints to the UML artifacts.
2. Generating Markov chain usage model from the UML artifacts with statistical testing constraints.
3. Testing the software based on the Markov chain usage model and evaluating the software reliability.

The software Markov chain usage model is a 5-tuple:

$$MC = (S, \Sigma, \delta, q_0, F)$$

where

- $S$  is the set of software execution states.
- $\Sigma$  is the set of transition labels. A label is of the form of  $\langle \text{event} \mid \text{action}, \text{Name}, \text{Probability} \rangle$ .
- $\delta$  is transition relation:  $\delta: S \times \Sigma \rightarrow S$ .
- $F$  is the set of software final states.
- $q_0$  is the initial state of software execution.

A state  $s \in S$  is defined as:

$$s = (State_{num}, Label, Notation)$$

where

- $State_{num}$  is the state number.

- $Label \in \{ \text{"Time violation"}, \text{"Delay"}, \text{"Initial"}, \text{"Final"} \}$ , which is used to distinguish initial state, final state and time related states.
- $Notation$  is the state's accessory information, for example, the postcondition of a sequence diagram's execution, or the state's delay time interval.

#### 3.2. Introducing test constraints to UML artifacts

We introduce the following constraints to UML models with OCL, which are indispensable for real-time software usage models generation:

- Preconditions that must be satisfied before a use case executes.
- Postconditions that the software enters after the execution of sequence diagrams.
- Execution probability of each sequence diagram in its associated use case.
- Execution sequence relations between use cases, which is depicted by a transition table or a UML activity diagram.
- Timing constraints attached with the send or receive events of the messages associated with the objects under testing (OUTs). A timing constraint is defined in section 2.3, which specifies the timing constraint that two consecutive events of the same instance must satisfy, time violation handler, execution postcondition of time violation handler, and time violation probability in the scenario.

Those constraints can be introduced during system analysis and design process, and will greatly increase testability of the software, facilitate understanding of the software system without too much modification to UML artifacts, thus can be easily accepted by system analysts and designers.

#### 3.3. Generating use case's usage model

To generate a Markov chain usage model of a UML use case with statistical testing constraints, we assume that the messages associated with the OUTs are totally ordered. The generation includes the following steps:

Step 1: Generating Markov chain usage model for the first sequence diagram of the use case. This procedure is depicted in Figure 9.

Step 2: Integrating other sequence diagrams of the use case into the usage model and thus generate the use case's usage model. Figure 10 shows this procedure.

Figure 11 is the procedure of processing each timing constraint specified with OCL in UML artifacts.

### 3.4. Generating software usage model

The algorithm in Figure 12 shows how to integrate the Markov chains of use cases to generate the entire software usage model. The algorithm checks each use case's precondition set and the final states of use cases' Markov chain. If the final state of use case *SourceUC*'s Markov chain,  $S_{out}$ , exists in use case *TargetUC*'s precondition set, which means that use case *TargetUC* will execute after use case *SourceUC*. Thus, a null transition  $t$  is created.  $S_{out}$  and  $S_{in}$ , which is the initial state of *TargetUC*, is linked by  $t$ . In fact,  $t$  reflects the use cases' execution sequence relation of use case *TargetUC* and *SourceUC*. Note that  $q_0$  is the initial state of software's Markov chain.

### 3.5. Discussion and case result

It is worth noting that different constraints added to UML artifacts may result in different methods to derive usage models of use cases. For example, if each message

in sequence diagrams is attached with a precondition and a postcondition, a more elaborate usage model can be generated. But, too much testability constraints may prevent the method from industrial acceptance. Weak constraints may produce redundant states and transitions, but since we are not concerned with deriving accurate software state models, but usage models, thus easy to use is the main concern.

It is well known that the transition probabilities in Markov chain usage models are hard to acquire because the granularity of transition is too small, which makes the transition probabilities difficult to sample and record. We utilize the execution probabilities of scenarios in their associated use case, which is relatively easy to acquire, and assign the execution probability of the scenario to the messages of this scenario. The probability of each message is mapped to each transition and thus, we get the transition probability of each transition.

Figure 13 shows the Markov chain of SCS generated by the algorithms presented in this section. Those states labeled with numbers are explained in table 2.

---

```

Create a Markov chain with initial state  $S_0$ ;
   $CurrentState \leftarrow S_0$ ;  $pf \leftarrow$  current scenario execution probability;

FOR each message  $m$  in current scenario associated with the OUTs
  IF message  $m$ 's event is attached with timing constraint THEN
    TimeTranslate ( $TC$ ,  $TmViolationProb$ ,  $TmViolationPost$ ,  $TmViolationHandler$ );
     $t \leftarrow \langle \text{event}, \text{null}, pf * (1 - TmViolationProb) \rangle$ ;
    Insert a state  $S$  and let the label of  $S$  be "Delay" and notation be  $TC$ ;
    link  $CurrentState$  with  $S$  by transition  $t$ ;
    Insert a state  $S_1$  and let the label  $S_1$  be "Time violation" and notation be NOT( $TC$ );
     $t \leftarrow \langle \text{event}, \text{null}, pf * TmViolationProb \rangle$ ;
    link  $CurrentState$  with  $S_1$  by transition  $t$ ;
    Insert a state  $S_2$  and let the label  $S_2$  be "Final" and notation be  $TmViolationPost$ ;
    link  $S_1$  with  $S_2$  by transition  $\langle \text{action}, TmViolationHandler, pf * TmViolationProb \rangle$ ;
     $CurrentState \leftarrow S$ ;
  ENDIF

  /* Translate the message to transition */
  IF  $m$  is the OUTs received message THEN
     $t \leftarrow \langle \text{event}, m, pf \rangle$ ;
  ELSE /*  $m$  is message the tested objects send */
     $t \leftarrow \langle \text{action}, m, pf \rangle$ ;
  ENDIF
  Insert a new state  $S$  and link  $CurrentState$  with  $S$  by transition  $t$ ;
   $CurrentState \leftarrow S$ ;
ENDFOR
Let  $CurrentState$ 's label be "Final" and notation be postcondition of current scenario.

```

---

Figure 9. Algorithm for translating the first sequence diagram into a Markov chain

---

```

FOR all other scenarios
  CurrentState  $\leftarrow S_0$ ;
  postC  $\leftarrow$  postcondition of current scenario;
  pf  $\leftarrow$  current scenario execution probability;

FOR each message m in current scenario associated with the OUTs
  IF m is the OUTs received message THEN
    tran  $\leftarrow$  < event, m, pf >;
  ELSE /* m is message the tested objects send */
    tran  $\leftarrow$  < action, m, pf >;
  ENDIF
  IF message m's event is attached with timing constraint THEN
    TimeTranslate (TC, TmViolationProb, TmViolationPost, TmViolationHandler);
    IF there is no such a state whose label is "Delay" and notation is TC in the
      successive state set of CurrentState THEN
      Create a state S and let its label be "Delay" and notation be TC;
      t  $\leftarrow$  <event, null, pf*(1- TmViolationProb) >;
      Insert S into the chain and link CurrentState with S by t;
      Insert a state S1 and let S1's label be "Time violation" and notation be NOT(TC);
      t  $\leftarrow$  < event, null, pf*TmViolationProb >;
      link CurrentState with S1 by transition t;
      Insert a state S2 and let S2's label be "Final" and notation be TmViolationPost;
      link S1 with S2 by transition <action, TmViolationHandler, pf*TmViolationProb >;
    ELSE
      Let S be the state whose label is "Delay" and notation is TC;
      t  $\leftarrow$  the transition from CurrentState to S;
      Accumulate pf*(1-TmViolationProb) to the transition probability of t;
      S1  $\leftarrow$  CurrentState's successive state whose label is "Time violation" and notation "NOT(TC)";
      t  $\leftarrow$  the transition that links CurrentState and S1;
      Accumulate pf*TmViolationProb to the transition probability of t;
      IF the state with notation TmViolationPost is the successive state of S1 THEN
        t  $\leftarrow$  the transition that links S1 and state with label "Final" and notation TmViolationPost;
        Accumulate pf*TmViolationProb to the transition probability of t;
      ELSE
        Insert state S2 and let S2's label be "Final" and notation be TmViolationPost;
        link S1 with S2 by transition <action, TmViolationHandler, pf*TmViolationProb >;
      ENDIF
    ENDIF
    CurrentState  $\leftarrow$  S;
  ENDIF
  IF (a transition tr exists in the chain which is same as tran except pf) AND
    (the source state of tr is CurrentState) THEN
    Accumulate pf to the transition probability of tr;
    CurrentState  $\leftarrow$  target state of tr;
    IF CurrentState's label is "Final" AND m is not current scenario's last message THEN
      /* Avoid cycles on the final states of the Markov chains of use cases*/
      insert a new state S and let S be the target state of tr;
      t  $\leftarrow$  <event, null, (transition probability of tr) - pf>;
      Link state S with CurrentState by transition t;
      CurrentState  $\leftarrow$  S;
    ENDIF
  ELSE
    Insert a new state S and link CurrentState with S by transition tran;

```

```

        CurrentState  $\leftarrow S$ ;
    ENDIF
ENDFOR

IF postC exists in the notations of the current chain's final states set THEN
    Unify CurrentState with the final state whose notation is postC;
ELSE
    Let CurrentState's label be "Final" and notation be postC;
ENDIF
ENDFOR
Normalize transition probability of each transition.

```

Figure 10. Algorithm for integrating other scenarios into usage model of a use case

```

PROCEDURE TimeTranslate (TC, TmViolationProb, TmViolationPost, TmViolationHandler)
    IF timing constraint is ( $t_e + a < t_e' < t_e + b$ ) THEN
        TimeConstraints  $\leftarrow (a, b)$ ;
    ENDIF
    TmViolationProb  $\leftarrow$  Time violation probability of timing constraint;
    TmViolationPost  $\leftarrow$  Time violation handler's execution postcondition;
    TmViolationHandler  $\leftarrow$  Time violation handler;
ENDPROCEDURE.

```

Figure 11. Algorithm for processing a timing constraint

```

Insert the first state q0 and let q0's label be "Initial";
FOR each use case TargetUC
    IF TargetUC is s_start's successive use case in UCExecRelation THEN
        pf  $\leftarrow$  the transition probability from s_start to TargetUC in UCExecRelation;
        S_in  $\leftarrow S_0$  of TargetUC's Markov chain;
        t  $\leftarrow \langle \text{event}, \text{null}, \text{pf} \rangle$ ;
        Link q0 with S_in by transition t;
    ENDIF
ENDFOR
FOR each use case SourceUC
    FOR each final state S_out in SourceUC's Markov chain
        FOR each use case TargetUC
            IF S_out  $\in$  precondition set of TargetUC THEN
                Find function seq(SourceUC, notation of S_out, pf) = TargetUC in UCExecRelation;
                t  $\leftarrow \langle \text{event}, \text{null}, \text{pf} \rangle$ ;
                S_in  $\leftarrow S_0$  of TargetUC's Markov chain;
                Link S_out with S_in by transition t and clear the label of S_out;
            ENDIF
        ENDFOR
        IF seq(SourceUC, notation of S_out, pf) = s_end in UCExecRelation THEN
            Insert a state s, let the label of s be "Final" and clear the label of S_out;
            t  $\leftarrow \langle \text{event}, \text{null}, \text{pf} \rangle$ ;
            Link S_out with s by transition t;
        ENDIF
    ENDFOR
ENDFOR

```

Figure 12. Algorithm for generating the software usage model

Table 2. Contents of the numbered states

State <sub>num</sub>	Label	Notation
1	Initial	
2	Delay	(3, 5)
3	Time violation	NOT (3, 5)
4	Delay	(3, 5)
5	Time violation	NOT (3, 5)
6	Final	
7	Delay	(3, 5)
8	Time violation	NOT (3, 5)
9	Final	
10	Time violation	NOT (3, 5)
11	Delay	(3, 5)
12	Final	

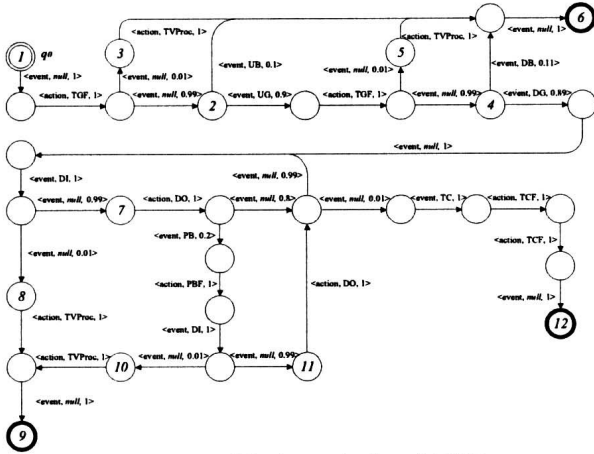


Figure 13. Markov chain of SCS

## 4. Test Case generation and automation

### 4.1. Test case generation

Statistical test cases are generated automatically by traversing the transitions of usage model from state labeled “Initial” to state labeled “Final”, guided by the probabilities associated with the outgoing transitions of each state [18]. Each transition of the usage model represents a particular stimulus or response of the software, thus the transitions can be classified into ‘event’ type or ‘action’ type. While traversing the usage model derived with the algorithms presented in this paper, not only the transitions are included to the traverse path but also the states labeled “Delay” or “Time violation” because they represent the timing constraints that the system must maintain. When they are included to the traverse path, the states whose labels are “Delay” and notations are (a, b) are translated to (delay(a, b)), and the states whose labels are “Time violation” and notation NOT(a, b) are translated to (delay(x)), in which x is a number no less than b; or

(delay(x)).(AnyStimulus), in which x is a number no greater than a. Thus, delay(3, 5) means that after the last interaction, the software may receive the next stimulus or send the next response in a random time between 3 to 5 time units. And delay(6) means the software delays for 6 time units. A test case is a path from the state labeled “Initial” to a state labeled “Final”.

Statistical testing continues until statistical testing adequacy criteria are satisfied [18]. If the testing cost and project schedule are main concerns, testing continues until test resources available are used up.

Take the example of SCS, the following testing case denotes a successful communication process:

```
[TGF] . (Delay(3, 5)) . (UG) . [TGF] .
(delay(3, 5)) . (DG) . (DI) . (delay(3, 5)) .
[DO] . ... . (DI) . (delay(3, 5)) . [DO] . (TC) .
[TCF] . [TCF]
```

The following test case denotes a failed communication because timeout occurs while making connection.

```
[TGF] . (delay(3, 5)) . (UG) . [TGF] .
(delay(6)) . [TVProc]
```

In a test case, (.) is the stimulus to the software and [.] is response of the software. The stimuli are translated into the testing scripts to simulate the user’s input to the software and responses can be used as the partial test oracle.

### 4.2. Automation

We have developed the tool RT-UMGen to support statistical testing automation. RT-UMGen supports test constraints adding and Markov chain usage models generation from the constrained UML artifacts.

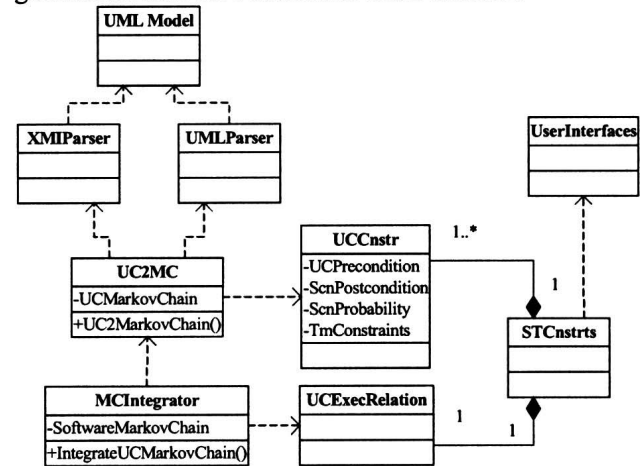


Figure 14. Class diagram of RT-UMGen

Figure 14 is the framework of RT-UMGen, where XMLParser accesses the UML model data by parsing XMI files; UMLParser accesses the UML model by APIs of the UML modeling tools. STCnstrts are the statistical

testing constraints, which includes use cases' constraints UCCnstr and use cases' execution sequence relations UCExecRelation. UC2MC derives use cases' Markov chain usage models, which are integrated by MCIntergrator, thus the software Markov chain usage model is generated. UserInterface provides the interaction interfaces to testers.

## 5. Conclusion and future work

This paper presents a method to generate the usage model from real-time software UML models. Compared to the method presented in [19], the distinguished feature is the incorporation of timing constraints in sequence diagrams. With the additional descriptions of timing constraints and statistical testing constraints in UML models, the method can generate the corresponding Markov chain usage models of the software systems, and facilitate real-time software statistical testing.

The ongoing and future work includes incorporating more information from UML artifacts, for example, class diagrams, deployment diagrams and performance and QoS requirements to generate usage models, thus provide the criteria for selecting test case data in the application domain and generating statistical test cases for distributed software.

## Acknowledgement

This work is supported by National Natural Science Foundation of China Grants No. 90104007 and No. 60233020, 863 Hi-Tech Program of China Grants No. 2001AA113202 and No. 2001AA113190, and Huo Ying Dong Education Foundation Grant No. 71064.

## References

- [1] H. Ben-Abdallah and S. Leue, "Timing Constraints in Message Sequence Chart Specifications", *Proceedings of the 10<sup>th</sup> International Conference on Formal Description Techniques FORTE/PSTV'97*, Osaka, Japan, 1997.
- [2] A. Abdurazik and J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", UML'00, LNCS 1939, Springer Verlag, 2000, pp. 383-395.
- [3] R. Alur, G.J. Holzmann, and D. Peled, "An analyzer for message sequence charts", TACAS'96, LNCS 1055, Springer Verlag, 1996, pp. 35-48.
- [4] R. Alur, K. Etessami, and M. Yannakakis, "Inference of Message Sequence Charts", *Proceedings of 22<sup>nd</sup> IEEE International Conference on Software Engineering*, 2000, pp. 304-313.
- [5] F. Basanieri and A. Bertolino, "A Practical Approach to UML-Based Derivation of Integration Tests", *Proceedings of the 4<sup>th</sup> International Software Quality Week Europe*, Brussels, Nov. 2000, pp. 20-24.
- [6] Binder, R., *Testing Object-Oriented Systems*, Addison-Wesley, 1999.
- [7] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing", *Software and System Modeling*, Vol. 1, 2002, pp. 10-42.
- [8] Bruegge, B. and A.H. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall, 2000.
- [9] Douglass, B.P., *Real-Time UML*, Addison-Wesley, 2000.
- [10] D. Harel and W. Damm, "LSCs: Breathing Life into Message Sequence Charts", *Proceedings of the 3<sup>th</sup> IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999, pp. 293-312.
- [11] ITU, *Message Sequence Charts*, Recommendation Z.120, ITU Telecomm. Standardization Sector, 1996.
- [12] X. Li and J. Lilius, "Timing Analysis of UML Sequence Diagrams". UML'99, LNCS 1723, Springer Verlag, 1999, pp. 661-674.
- [13] Prowell, S.J., C.J. Trammell, R.C. Linger, and J.H. Poore, *Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999.
- [14] OMG, *Response to the OMG RFP for Schedulability, Performance, and Time*, OMG Document number: ad/2001-06-14, June, 2001.
- [15] M. Riebisch, I. Philippow and M. Götze, "UML-Based Statistical Test Case Generation", LNCS 2591, Springer, 2003, pp. 394-411.
- [16] Rumbaugh J., I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.
- [17] Warmer, J. and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.
- [18] J.A. Whittaker, "A Markov Chain Model for Statistical Software Testing", *IEEE Transactions on Software Engineering*, Vol. 20, Oct 1994, pp. 812-824.
- [19] J. Yan, J. Wang, and H. Chen, "Deriving Software Statistical Testing Model from UML Model", *Proceedings of QSIC 2003*, Dallas, USA, 2003, pp. 343-350.